1. Structured Programming
   - Principles of software engineering
   - Writing structured code in a procedural language

2. Object Oriented Programming
   - Principles of object-oriented design and programming
   - Writing object-oriented code

3. Computer Communications and Networking
   - How devices and computers communicate, the Internet
   - Writing programs that can exchange information on a network

4. Operating Systems
   - Principles of hardware abstraction and virtualisation
   - Writing programs that access hardware features transparently and concurrently

**Meta Learning Outcomes**

- Understand the importance of good design practice in software, and the role that structured and object oriented programming ideas play in this.
- Understand how to develop applications that can interact with the outside world (i.e., programs as part of bigger engineering systems) via the operating system and via inter-computer and computer-device communications.
- Understand how ideas in 4th year options and projects can actually be implemented in software.

**The First of the Four Courses**

- Software engineering principles
  - Mostly about concepts/principles of design, modularity, abstraction, encapsulation, etc.
- Structured programming
  - Revision, coding in C and MATLAB
  - Control flow, variables, data types
    - How variables are implemented in hardware
  - Functions
    - Code re-use, parameters, libraries
  - Data structures
    - Structures/classes, arrays
- Algorithms
- Recursion
Learning Outcomes

The course will aim to give a good understanding of basic software design methods, and emphasise the need to produce well-structured maintainable computer software. The course will concentrate on principles, but these will be reinforced with examples in MATLAB and C/C++ programming languages. Specifically, by the end of this course students should:

- Understand concepts of basic program design techniques that can be applied to a variety of programming languages.
- Understand the need for structured programming in software projects.
- Understand the mechanics of function calls and of recursion.
- Understand the role, uses and advantages of compound data structures.
- Be able to recognise, produce and/or maintain well structured programs.

Lecture 1 Outline

- Importance and challenges of software engineering
  - Examples & challenges

- Software engineering
  - What it shares and how it differs from other engineering
  - Abstraction and modularity

- Software processes & their models
  - Specification, design & implementation, validation, evolution
  - Waterfall and incremental models

- Structured programming
  - Key ingredients
  - Language support

- Design tools
  - Flow charts, pseudo-code, data flow diagrams, state diagrams

Texts

- The C programming language, 2nd edition
  - Kernaghan & Ritchie

- Introduction to algorithms
  - Cormen, Leiserson, Rivest, Stein
Computing is ubiquitous in engineering. Why?

- Computers support complex analysis and simulations
- In design and modelling:
  - Computer-assisted design reaches far beyond manual design
  - Many modern design techniques are infeasible without computers
- Awesome applications:
  - Embedded control: low power, fast, flexible
  - Personal "computing" devices: laptops, mobile phones
  - Digital economy: search, manipulate, find value in big (world-scale) data (Google)
- Cheap to reproduce and deploy
- Software is the key!

Some examples ...

### Design & Control: Sizewell B
- Nuclear power station (PWR), on-stream in 1995
- Software used extensively in the design
- Software used for control

### Design & Control: A380
- 1400 separate programs
- There is a software project just to manage all the software!
- Clearly safety-critical features of the software

### Design & Control: E- and S- Class 2013
- Extensive assisted driving support
- Emergency breaking (pedestrians, front & lateral cars)
- Smart cruise control (follow lanes, maintain distances, spots holes/bumps adapting suspensions)
- Parks automatically
Pre-crash braking
Body control
Parking
Adaptive cruise control
Lane keeping
Traffic signs
Adaptive high beam
Night view
Attention

[Pre-crash braking, Body control, Parking, Adaptive cruise control, Lane keeping, Traffic signs, Adaptive high beam, Night view, Attention]

Big Data: NPfIT

- **NHS National Plan for IT**
- **Goal** provide electronic care records for patients
  - Connect 30000 GPs and 300 hospitals
  - Provide secure access to records for healthcare professionals
  - Provide access for patients to their own records via “Healthspace”

The “size” of software

**SLOC**: number of source lines of code

```
#include "fisher.h"
#include "gmm.h"
#include "mathop.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

static void
VL_XCAT(vl_fisher_encode_, SFX)
(TYPE * enc,
 TYPE const * means, vl_size dimension, vl_size numClusters,
 TYPE const * covariances,
 TYPE const * priors,
 TYPE const * data, vl_size numData,
 int flags)
{
 vl_size dim;
 vl_index i_cl, i_d;
 TYPE * posteriors ;
 TYPE * sqrtInvSigma;

 posteriors = vl_malloc(sizeof(TYPE) * numClusters * numData);
 sqrtInvSigma = vl_malloc(sizeof(TYPE) * dimension * numClusters);

 memset(enc, 0, sizeof(TYPE) * 2 * dimension * numClusters);

 for (i_cl = 0 ; i_cl < (signed)numClusters ; ++i_cl) {
  for(dim = 0; dim < dimension; dim++)
   { sqrtInvSigma[i_cl*dimension + dim] = sqrt(1.0 / covariances[i_cl*dimension + dim]);
  }

 VL_XCAT(vl_get_gmm_data_posteriors_, SFX)(posteriors, numClusters, numData,
 dlflags).
```

About 50 SLOCs worth of code
Intrinsic difficulties with software

- **Complexity**
  - Handled via modularity

- **Invisible interfaces**
  - Modularity is tricky due to subtle and invisible interactions between software components (esp. concurrency)

- **The “curse” of flexibility**
  - Can encourage unnecessary complexity
  - Redefinition of tasks late in development – shifting goal-post

- **Analog vs discrete**
  - Analysis of analog systems can often be interpolated (e.g. if a bridge stands a certain load, it will stand smaller ones)
  - Not so for software

- **Historical usage information**
  - Unlike physical systems, there is a limited amount of experience about standard designs

---

When software projects go wrong

- **A320, Habsheim (1988) and Strasbourg (1992)**

- **London Ambulance Service**
  - 1992, computerised ambulance despatch system fails

- **Therac-25**
  - 2 people died and several others exposed to dangerous levels of radiation because of software flaws in radiotherapy device

- **OSIRIS**
  - £5M University financial package
  - Expenditure to date more like £20-25M

- **NPfIT?**
  - NHS £12B IT project

- **More**
  - `comp.risks` is a great source of others...
NHS National Programme for IT: NPfIT

- Plan to provide electronic care records for patients
- Connect 30000 GPs and 300 hospitals
- Provide secure access to records for healthcare professionals
- Provide access for patients to their own records via “Healthspace”

Laudable?

Realistic?
- Software Engineering specialists have their doubts
- Ross Anderson (Prof of Security Engineering, Cambridge Computing Laboratory) writes in his blog “I fear the whole project will just continue on its slow slide towards becoming the biggest IT disaster ever”.

Software Engineering

- Software Engineering
  - more than just programming/coding.
  - concerned with all aspects of software production.
- It seeks principles and methodology to make programs that are
  1. usable
     - meet their requirements, are acceptable by the users
  2. dependable
     - reliable, secure, safe
  3. maintainable
     - can be updated with minimal effort
  4. efficient
- Includes
  - theory of computation
  - programming tools (e.g. languages)
  - design tools (e.g. software architectures)
  - processes of software creation & management
  - ...

Software vs “Other” Engineering

- How is software engineering similar to other engineering?
- Modularity and Abstraction
  - Modularity: decompose a system into components
  - Abstraction: separate a component behavior from its implementation
- Benefits of modularity and abstraction
  - better understanding
  - knowledge reuse
  - isolate local changes
- Examples
  - Thevenin / Norton equivalent of a linear circuit
  - Transistors, flip-flops, registers, execution pipelines, chips, computers
Abstraction and modularity

$$V_o = A(V_+ - V_-)$$

Software vs “other” engineering

- How is software engineering different from other engineering?
- **Software products**
  - can incorporate massive complexity
  - can be easily modified
  - are weightless
  - do not age, corrode, etc.
  - do not have manufacturing defects
- **Cost of software**
  - cheap to manufacture (transfer a file)
  - expensive to produce (design, implement, validate, run, maintain, evolve)
- Vs computer hardware engineering
  - computer hardware engineering ≠ software engineering
  - however, software must account for hardware limitations

Lecture 1 Outline

- Importance and challenges of software engineering
  - Examples & challenges
- **Software engineering**
  - What it shares and how it differs from other engineering
  - Abstraction and modularity
- **Software processes & their models**
  - Specification, design & implementation, validation, evolution
  - Waterfall and incremental models
- Structured programming
  - Key ingredients
  - Language support
- **Design tools**
  - Flow charts, pseudo-code, data flow diagrams, state diagrams

Software processes

An **software process** is a set of related activities that leads to the production of a software product [Sommerville].

- **Fundamental activities (lifecycle)**
  1. Specification
  2. Design & implementation
  3. Validation
  4. Evolution
**Specification**

The process of understanding and defining what services are required from the system and identifying the constraints on the system’s operation and development [Sommerville].

- Requirements must be elicited, specified, and validated
- Identify services and constraints by talking to the user
- Refine these iteratively
- Produce a requirement specification (document)

**Example**

- User requirement: “calculate the n-bounce trajectory of a lossy bouncing ball”
- Must be refined:
  - How will the output be represented?
  - How will the initial condition be specified?
  - Which approximations are tolerated in the numerical integration?
  - Should the air resistance be accounted for?
  - ...

**Design & implementation**

**Design**

- Modules
  - architecture (which components)
  - interface (how components inter-operate)
  - components (how components work)
- Database
- User interface
- ...

**Implementation**

- Programming each component
- Unit testing & debugging
- Fine-grained design decisions are left to the programmer

**Validation**

- **Top-down design**
  - Keep in mind the general principles:
    - Abstraction
    - Modularity
  - Architectural design
    - identifying the building blocks (modules)
    - describe the data, functions, and their constraints
  - Interfaces
    - define how the modules fit together
  - Component design
    - recursively design each block

- **Verification**
  - Verification: does the software conform to its specifications?
- **Validation**: does the software actually do what it was supposed to do?

**Designing tests** for verification

- **Black-box** (from software specification) vs **white-box** (from code inspection)
- **Top-down** (system testing) vs **bottom-up** (unit testing)

**Coverage of the tests**

- Exhaustive testing is impossible
- Pick representative examples of “normal” inputs as well as “corner cases”

**Example**: Test a function to compute the tangent
- normal input: \( \tan(1.1) \)
- corner cases: \( \tan(-\pi/2) \), \( \tan(0) \), \( \tan(\pi/2) \)
Software evolution may be triggered by changing business requirements, by reports of software defects, or by changes to other systems in a software system’s environment [Sommerville].

- **Example goals** of software evolution
  - fix a bug
  - add a functionality
  - improve efficiency

- Evolving software is costly
  - require a good understanding of the system, which often must be re-acquired
  - effects of changes are difficult to predict and risky

- **Reengineering** may be needed to accommodate changes
  - **refactoring**: rewrite part of it with modern tools, a better design, etc
  - rewrite the documentation
  - sometimes rewrite from scratch (but keep the functionality equivalent)

---

**Waterfall model**

![Waterfall model diagram]

**Fundamental activities**
1. Specification
2. Design & implementation
3. Validation
4. Evolution

- Ideally, activities are distinct stages to be signed off chronologically
- In practice, activities partially overlap and interact

---

**Incremental models**

- Develop the software by increments, exposing them to the user and elicit changes, and go back to incorporate them.

- **Extreme programming**
  - interleave frequent software releases with user validation
  - development is test based
    - tests encode the specification
    - tests are written before the program
    - all releases must pass all tests
    - do not try to predict changes; changes reactively (refactoring)

- **Extreme programming vs waterfall**
  - much more lightweight
  - usually preferable for small-medium size projects
Lecture 1 Outline

Importance and challenges of software engineering
Examples & challenges

Software engineering
What it shares and how it differs from other engineering
Abstraction and modularity

Software processes & their models
Specification, design & implementation, validation, evolution
Waterfall and incremental models

Structured programming
Key ingredients
Language support

Design tools
Flow charts, pseudo-code, data flow diagrams, state diagrams

Structured Programming

Modularity is recursive
Components are recursively decomposed into simpler components, until machine instructions are generated. There are two ways of constructing such a hierarchy:

Top-down implementation
Code high level parts using "stubs" with assumed functionality for low-level dependencies
Iteratively descend to lower-level modules

Bottom-up implementation
Code and test each low-level component
Need “test harness” so that low-level components can be tested in its correct context
Integrate components
Not hard-fast rules; combination often best

Structured Programming

Modules
Group related functionalities together
Clearly separate behaviour from implementation
encapsulation, information hiding, ...

Control flow
for, repeat, while
Go To Statement Considered Harmful [Dijkstra]
procedures (or functions)
e.g., sin(), addAppointmentToCalendar()

Data
compound type (e.g. struct)
objects
Most computer languages have features that help structured programming
However, it is fundamentally a matter of how you organise your code

Modularity: Algorithm and Data Structures

Algorithms + Data Structures = Programs

[Wirth 75]

Different languages encourage decomposing programs in different ways:

Procedural languages
decompose algorithms

Object-oriented languages
decompose data
Lecture 1 Outline

- Importance and challenges of software engineering
- Examples & challenges
- Software engineering
  - What it shares and how it differs from other engineering
  - Abstraction and modularity
- Software processes & their models
  - Specification, design & implementation, validation, evolution
  - Waterfall and incremental models
- Structured programming
  - Key ingredients
  - Language support
- Design tools
  - Flow charts, pseudo-code, data flow diagrams, state diagrams

---

Simple design tools

Flow charts & pseudo-code

```
count = 0
while (not ready() and count <= 3)
    count = count + 1
    Hit snoozer
    while count < 3
        if ready()
            Climb out of bed
    Count
    Stop
```

---

Simple design tools

Data flows

VTOL (Vertical Take-Off and Land)

- Controller
  - trust
  - state
  - Display

- Simulator
  - state
  - velocity
  - gravity

- Display
  - trust
  - height

Simple design tools

State Diagram

- Controller
- Simulator
- Display

- Trust
- State
- Velocity
- Gravity

- Operation
- Operation
- Operation
- Operation

- Door Open
- Door Closed
- Door Open
- Door Open

- Enabled
- Disabled
- Enabled
- Disabled

- Do: Set Power = 600
- Do: Set Power = 300
- Do: Display Ready
- Do: Display Waiting

[Sommerville]